

Augmenting In-Context-Learning in LLMs via Automatic Data Labeling and Refinement

Joseph Shtok, Amit Alfassy, Foad Abo Dahood, Eliyahu Schwartz, Sivan Doveh and Assaf Arbelle
Multimodal AI team, IBM Research AI *

Abstract

It has been shown that Large Language Models' (LLMs) performance can be improved for many tasks using Chain of Thought (CoT) or In-Context Learning (ICL), which involve demonstrating the steps needed to solve a task using a few examples. However, while datasets with input-output pairs are relatively easy to produce, providing demonstrations which include intermediate steps requires cumbersome manual work. These steps may be executable programs, as in agentic flows, or step-by-step reasoning as in CoT. In this work, we propose Automatic Data Labeling and Refinement (ADLR), a method to automatically generate and filter demonstrations which include the above intermediate steps, starting from a small seed of manually crafted examples. We demonstrate the advantage of ADLR in code-based table QA and mathematical reasoning, achieving up to a 5.5% gain. The code implementing our method is provided in the Supplementary material and will be made available.

1 Introduction

The past decade has seen a big renaissance in the Machine Learning (ML) domain with the rise of neural networks which continue to break all limits at a rapid pace. Until recently, the common training paradigm was based on task-specific models, each trained on a separate dataset for a given task, e.g. classification [Krizhevsky et al., 2012], detection [Redmon et al., 2016], summarizing [Nallapati et al., 2016], translation [Vaswani et al., 2017], etc. Today, we see the rise of Foundation Models [Bommasani et al., 2021] largely based on Large Language Models (LLMs), which have several interesting emerging properties, including In-Context-Learning (ICL) and Chain-of-Thought (CoT) inference. ICL is an approach where the model's behavior is modulated through the model's input, i.e. the context. This context can include information that is required to answer a desired query. This concept is extremely useful in several pipelines, for example in Retrieval-Augmented Generation (RAG) [Lewis et al., 2020] systems. In other cases, the context can include several examples of input-output pairs that outline the models' expected behavior. This property of the model of receiving the small training data for the required task *inline*, without modification of weights, was envisioned at the earlier stage of ML development as the concept of meta-learning [Vinyals et al., 2016] and implemented as few-shot learning, where few training examples are appended to the input query.

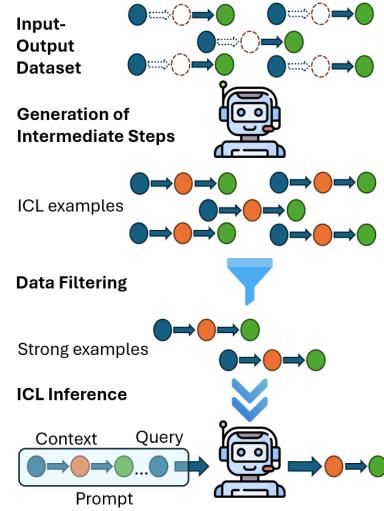


Figure 1: From an input-output dataset with no intermediate steps (CoT/Executable programs), ADLR generates examples with such steps and retains the useful examples. These examples are used in the ICL to answer a given query.

* corresponding author: Joseph Shtok, josephs@il.ibm.com

The ICL regime has proven advantageous for LLMs on general tasks [Liu et al., 2022b], but it is particularly strong when a task, assigned to the LLM, has a particular structure and presumes a specific form of response. In this scenario, the model’s input, a.k.a the prompt, consists largely of the context and is composed of several examples exhibiting the domain and format of the input data, as well as the nature of the required output. In fact, these examples contain the expert knowledge that defines the task. Examples of such special tasks include the CoT approach to solve complex tasks that require reasoning, such as Math Word Problems, with demonstrations of step-by-step mathematical arguments. Another example where ICL is successfully used is the neuro-symbolic, or agentic flow - approach which leverages code models to generate executable scripts, which in turn are executed to generate the desired answer [Gupta and Kembhavi, 2023; Surís et al., 2023].

One example is the Table QA task where answering the question requires complex logical and mathematical operations. When tackling this problem with ICL, the LLM’s prompt is designed to steer the network into solving the query through task-specific code, usually in Python or SQL. To that end, the prompt examples are designed to contain the table, the question, and the program that should run on the table to answer it. The aforementioned expert-knowledge examples are usually hand-crafted for the specific dataset in small amounts, and the same prompt, comprising of those examples, is used for all queries. This approach doesn’t leverage the large available training sets, containing diverse samples that can be used despite not being ready-made for the context.

This paper introduces a simple and useful method for improving the performance of existing algorithms operating LLMs in the ICL regime where intermediate steps, such as code generation or CoT is used. Specifically, we seek to augment the initial set of manually crafted few-shot examples with a much larger set of automatically computed examples which are shown to be (1) hard for the LLM to solve and (2) useful in solving other hard samples in the few-shot context.

There are then three steps we take towards our goal, as outlined in Figure 1. First, given a training dataset comprised of inputs and final answers, we execute the given algorithm using the initial hand-crafted context to generate the intermediate data for these samples. It’s correctness is verified by checking that it leads to the correct final answer (by continuing the generation, in case of CoT, or by executing the computer program, in case of code-based QA). This provides us both with a collection of full solved examples, and with a set of hard (unsolved) samples.

In the second step, we refine the collection of solved examples, using the two aforementioned criteria. The difficulty of a sample is estimated by the portion of LLM runs, with the same input and a non-zero temperature, that results in a correct solution. We restrict the pool of examples to the hard ones, as those possibly contain more new information for the LLM. Further, we subject these examples to the test of utility, seeking to choose those that have proven themselves able to solve many hard samples in a one-shot prompt. Finally, we employ the refined pool of examples to augment the inference protocol of the baseline algorithm. We use multiple diverse contexts, populated with random subsets of the pool, and containing a high number of examples. The results of the corresponding multiple runs of the LLM are aggregated via the majority vote (by the common practice).

The contributions of this paper are the following: [1] We propose a method for improved context generation in ICL-based LLM inference for complex tasks improving the current results by up to 5.5%; [2] We conduct an ablation study exhibiting different aspects of the ICL inference as well as those of our method; [3] We release an open-source code repository to make the method directly accessible by the community (please see details in Supplementary Section A).

2 Prior Art

2.1 Automatic prompt selection and generation for the In-Context Learning

There is good evidence that invoking ICL for the few-shot scenario can improve the performance of an LLM on a number of tasks [Brown et al., 2020]. In this approach, the context of the prompt includes the few-shot exemplars that guide the model in solving the task. Both the selection process itself and the construction of the prompt from the selected examples have been a subject of interest for many prior works. Initially, the few-shot examples were carefully and manually curated for every task. This manual process is neither scalable nor stable, as there is no guarantee that the manually selected examples are optimal for the task. A different approach is to select the few-shot exemplars from a large pool of samples with the common practice of randomly selecting the samples from the pool [Brown et al., 2020; Cheng et al., 2023; Zhang and Henkel, 2023]. However, it has been shown

that the ICL performance can be unstable since it is dependent both on the selection and ordering of the context examples [Zhang et al., 2022a; Gao et al., 2021; Liu et al., 2022b]. Some methods strive to stabilize the model by dynamically selecting the context examples depending on the given query. One selection method, used in [Liu et al., 2022b], is a similarity-based K-nearest neighbors (K-NN) selection, where the semantic similarity is computed using the embeddings of a different model, in this case a pre-trained RoBERTa model. In a number of other works, dedicated networks are trained to exercise the sample selection either via Reinforcement Learning in [Lu et al., 2022] and [Scarlato and Lan, 2023], or via contrastive learning in [Rubin et al., 2022]. In our work, we refrain from the ambitious goal of training a model to predict the usefulness of shots for LLMs with respect to a particular query; instead, we gather statistics (over a pool of candidate examples) of the usefulness of the examples from direct one-shot evaluations of the LLM. This results in a much simpler output - a collection of generally strong examples, proven to work for difficult queries. This approach has the added benefit of not requiring an execution of an additional model during inference.

Another direction of improving the sample selection is restricting the pool of examples to a high-quality subset, as was done in [Chang and Jia, 2023]. Its authors show that curation of the selection pool both stabilizes the ICL performance and improve it. Inspired by these considerations, we incorporate two steps of data filtration into our process, as will be detailed further in Section 3.

Avoiding the use of the small set of manually crafted expert examples was also considered in [Zhang et al., 2022b], which addresses the augmentation of Chain of Thought (CoT) for complex reasoning. The authors demonstrate that reasoning chains, generated with an LLM by a corresponding instruction, can successfully replace the hand-made chains, given enough diversity. For the selection of “good” samples for the prompt, the authors used semantically-based clustering. In contrast to this work, we extend the hand-crafted examples, rather than replace them, by generating *verified* demonstrations, as detailed in Section 3.

2.2 Table QA with code LLMs

There are two approaches to addressing the limited reasoning and mathematical skills of LLMs for the table QA task. One way is to fine-tune a general-purpose LLM on data involving operations of this kind [Zhao et al., 2022; Liu et al., 2022a; Jiang et al., 2022; Patnaik et al., 2024]; the other approach is to employ a code-generating LLM in writing an executable program which performs the necessary operations on the table [Ni et al., 2023; Ye et al., 2023; Cheng et al., 2023; Zhang and Henkel, 2023; Wang et al., 2024]. Our work belongs to this second group.

Since answering questions on tables may require non-trivial mathematical operations such as condition-based filtering, $\max()$, $\sum()$ and others, LLMs are not always able to directly answer the queries. Therefore, most leading SoTA solutions for table QA were built on strong off-the-shelf code LLMs², producing the final answer by executing the generated code on the input table. Binder [Cheng et al., 2023] introduced the approach of generating programs with embedded API calls to an external expert to resolve the difficulties of interpreting the question in terms of table headers and rows. Lever [Ni et al., 2023] introduced an orthogonal feature of a trained model verifying and selecting the output programs from multiple executions of the LLM on the same query, to retain those more likely to produce the correct answer. Dater [Ye et al., 2023] took yet another approach of helping the LLM to interpret the question by breaking down the process into intermediate tasks.

Chain-of-Table [Wang et al., 2024] shift the focus of CoT concept from the logic to the data, gradually updating the table in order to help interpreting the question in terms of its content. ReActTable [Zhang and Henkel, 2023] employ the ReAct paradigm [Yao et al., 2022] to progressively enhance the input data by generating, similarly to Chain-of-Table [Wang et al., 2024], intermediate data representations and injecting the results of the code interpreter back to the LLM querying process. Most of these works could benefit from an orthogonal feature of having a stronger pool of few-shot examples, instead of the few hand-crafted ones.

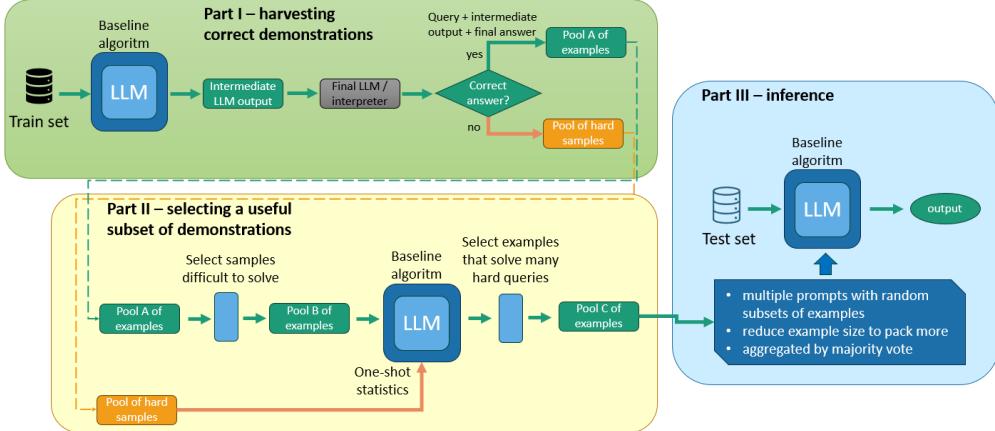


Figure 2: Flow diagram of ADLR

3 The Automatic Data Labeling and Refinement (ADLR) method

As was described in Section 1, our method consists of three parts (presented in the diagram in Figure 2): [1] Produce a large pool of samples annotated with the desired intermediate outputs, which can serve as the few-shot examples (demonstrations); [2] filter and refine the original pool to contain a subset of high-quality examples; and [3] perform ICL with the selected examples, using a protocol involving a large set of diversified prompts.

These steps are orthogonal to any particular algorithm using ICL. Parts 1,2 are applied while using the underlying algorithm as a black-box, while the third one requires minimal altering to the algorithm’s inference protocol.

3.1 Part I - generating a large pool of examples

The first step is to extend the initial hand-crafted set of examples by using the large training set, that does not have the desired intermediate outputs but does include the final answers. The task is akin to classical semi-supervised learning (SSL), where there are few annotated samples (in our case, the hand-crafted ones), and a lot of unlabeled. But in contrast to SSL, where there is no possibility to verify the pseudo-labels produced for unlabeled data, we can use the final answers. In the case of Table QA, these are the answers to the table questions, obtained after running the generated code; in the case of CoT, there are the expected final responses of the LLM after going through the chain of thought. So we make here the non-rigorous, but statistically plausible assumption that if the intermediate generated data leads to the correct answer, it is by itself correct. Thus we simply accept the intermediate data to build an example, if the final answer is correct.

By running the inference of the baseline algorithm on the training data, we thus obtain a large pool of fully solved examples - e.g., for WikiTQ, where the performance stands on 60% – 68% accuracy, this yields a pool of examples of size at least $\sim 60\%$ of the training data. We refer to the obtained pool of samples as “Pool A”.

3.2 Part II - focusing on the useful examples

On our way to creating strong few-shot prompts, we seek to single out the examples that would help the LLM produce the desired results. This, in general, is a difficult problem, and is approached in the literature by retrieval of examples for a given query (using semantic similarity or a pretrained network, as discussed in Section 2). We take a different approach and produce a general set of useful examples, which are randomly sampled to create multiple diverse prompts, as detailed later.

²With the exception of the work [Patnaik et al., 2024] from ICLR 2024, which achieves SOTA with a finetuned LLM.

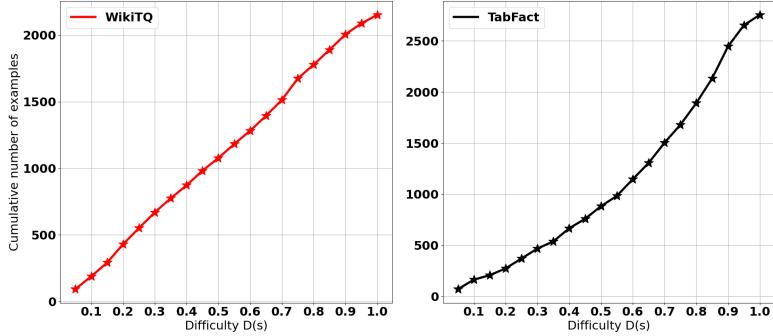


Figure 3: Cumulative distribution of pool-A examples according to the sample difficulty metric $D(s)$

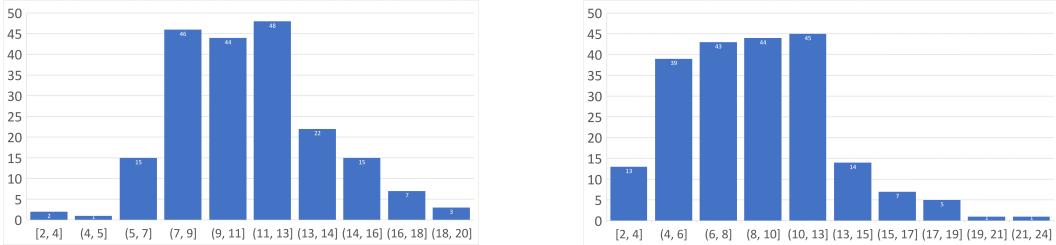


Figure 4: Histograms of success rate (percents of solved samples) for pool-B examples, applied in one-shot regime

First, we look closer into the LLM inference protocol in tasks like table QA: by common practice the LLM is executed N times (with $N \sim 10$) with the sample prompt, while setting the LLM temperature at non-zero value, and the various answers are aggregated via the majority voting. This technique improves the accuracy and stability of the system [Xuezhi Wang et al., 2023]. Also, it enables us to introduce a metric for the level of difficulty of the samples: for a sample s , on which $k \leq N$ executions of the LLM have led to the correct result we define the difficulty $D(s) = \frac{k}{N}$.

In Figure 3 we plot the number of successfully solved training samples, as a function of the *maximal* number k of attempts the LLM has succeeded for them, for $N = 20$ attempts. Thus, in the case of WikiTQ dataset, out of 3500 used train samples, 92 were solved with just one prompt out of 20, 188 were solved with either 1 or 2 attempts and so on. The takeaway from this analysis is the subset of samples that are, on one hand, solvable, but on the other hand, hard - $D(s)$ restricted from above. We set the threshold at 0.2, using the samples solved with at most 4 out of 20 attempts. These constitute the “pool B” of our protocol.

The second challenge, posed to our candidate examples, is their usefulness in few-shot scenarios. To gauge that, we run a massive set of experiments with one-shot prompts, populated with examples from pool B, and the queries are selected from the set of samples unsolved at the initial inference on the train data. Interestingly, percentages of those samples that do get solved when a “right” one-shot prompt is used, are quite high: 68.2% and 71.8% in WikiTQ and TabFact datasets, respectively. This tells us that improving the performance on the given data is well within the capabilities of the LLM, it just needs to receive the right examples in the context.

Now, we have per-example performance: how many of the hard queries was it able to solve. So, we further restrict the pool of examples to those that have solved at least 10% of the queries. In Figure 4 we present a histogram of the success rate of the pool B shots, where the 10% threshold was applied. On one hand, the threshold is high enough to retain only the strong examples, on other hand - there is still enough such examples (to the left of the “10” value in the histogram). These constitute the final “pool C”, employed for populating the prompts for ADLR. It is merged with the original, hand-made set of examples.

The work of our pipeline can be visualized via the amounts of samples from the training set that are gradually refined from pool A to pool C. The numbers are given in Table 1).

Table 1: Amounts of examples at different stages of selection, obtained for the Binder implementation

	Pool A	Pool B	Pool C	Unsolved Samples
WikiTQ	2057	429	135	1433
TabFact	2755	363	128	743

3.3 Part III - augmenting the inference protocol

With the restricted pool of few hand-crafted examples, inference protocols in most of the relevant algorithms comprise of (1) using a single context, composed of the available examples; (2) running the LLM on the prompt composed of this context and the query multiple times, with non-zero LLM temperature (0.5 – 1.0); (3) integrating the (possibly) different predictions among the successful attempts³ via a majority vote. This protocol performs better than a single-attempt run [Xuezhi Wang et al., 2023], both because the LLM has a better chance to produce a working code, and because the majority voting helps filtering out outliers.

One of the main features of our approach is the ability to produce multiple diverse contexts, by sampling of the pool of examples. We have experimented with similarity-based K-NN method Liu et al. [2022b], as detailed in Section 2.1, but eventually we chose to use random sampling, with no dependence on the query sample. Beyond that, we used the enlarged examples pool to increase the number of demonstrations in each prompt. This step is, of course, dependent on the size of the examples and the capacity of the employed LLM; in our experiments we used CodeLlama-34b and GPT-3.5-Turbo, that allow for 16k tokens as inputs.

4 Experiments

4.1 Experimental setup

Baseline methods and Datasets We evaluate our method on three widely used table QA datasets: WikiTableQuestions (WikiTQ) [Pasupat and Liang, 2015], Table Fact Checking (TabFact) [Chen et al., 2020], and the TabMWP [Lu et al., 2022] datasets. Since our proposed approach aims to improve the in-context example selection process, we restrict our comparisons to methods that use in-context examples that include intermediate outputs, i.e, either code-generation or CoT. For the first two datasets, where the task is Table QA, the evaluation is conducted by applying our method to two strong table QA solutions that use intermediate outputs in the form of code generation: the Binder [Cheng et al., 2023] and the ReacTable [Zhang and Henkel, 2023] algorithms. In both cases, we used the original code, shared by the authors, and wrapped it with our algorithm. The final results are presented in Table 2. On the TabMWP, we demonstrate the merits of our sample selection, continuing the study of different selection policies in the original [Lu et al., 2022] paper. Here, “Part I” of ADLR is not required, since the samples of the training set already contain the detailed “think step by step” thought process, leading to the answer, a solution of a math problem. However, as can be seen in Table 3 the following filtration of the pool of examples proves fruitful for final performance.

4.1.1 Selecting an LLM

While Binder has invoked the Codex (code-davinci-002) model by OpenAI, our reproduction (and augmentation) of its algorithm was done with the CodeLlama-34b [Rozière et al., 2023], an open-source model. While the reproduced baseline performance was somewhat lower than that reported in [Cheng et al., 2023], the positive outcome was the availability of a quality open-source solution for these two datasets (just 1.1% short of the performance attained with the proprietary engine, with our augmentation). In the case of ReAcTable, in which the GPT-3.5-Turbo was used, the substitution of LLM with open-source alternative have caused substantially worse results; the large performance gaps due to different LLMs are also observed in the literature [Lu et al., 2023]. The reason, we believe, lies in the fact ReAcTable requires more complex outputs, relying more on the LLM’s wits. This, in fact, is another hidden factor in choosing the right algorithm for a task solution, not discussed

³in code-based table QA, some runs may result in a technically incorrect code which fails to run at the interpreter

in the current literature: while some methods (like Binder) get somewhat lower performance than those relying on strong LLMs, they may be preferable in the sense that they rely on free models for the text generation and thus reducing the cost of operations. In light of these findings and since Codex is no longer available, we have reproduced the ReAcTable evaluation using GPT-3.5-Turbo. Additionally, for TabMWP, the selection of the LLM also plays an important role. As was shown in [Lu et al., 2023], GPT-4 of OpenAI can attain 91.81% accuracy in the few-shot regime in a plain application to this data (i.e., with the provided CoT inputs), while GPT-3 delivers only 62.92% in this regime. We again have experimented with some open-source LLMs, showing intermediate results; as our purpose here is to show a relative improvement (for different selection policies) using our proposed algorithm, the results are reported for the CodeLlama-34b [Rozière et al., 2023] and the Mixtral-8x7b-instruct [Jiang et al., 2024] models.

4.1.2 ADLR parameters and inference protocols

We summarize here the meta-parameters used in applying our method, including the parameters of its inference protocol, as well as the baseline inference protocol. Part I has no parameters to set - we simply run the baseline algorithm through the training set in order to harvest solved and unsolved samples. For the reasons of computation savings, we used only 3500 samples for each of the WikiTQ and TabFact datasets, and 8k samples from the TabMWP data. The reason for larger pool in TabMWP is that in the inference the samples are further partitioned into question types and answer types, and we need a sufficient amount of examples of each type to populate the contexts. Examples of TabMWP data, as well as details of questions/answers types, are provided in Supplementary material.

In Part II, pool B is produced as a subset of pool A of samples which are solved in at most 20% of LLM runs: $D(s) = \frac{k}{N} < 0.2$. Then in the one-shot statistics, each shot (A pool-B member) is selected for pool C if (1) it was used on at least 100 hard samples, and (2) it has solved at least 10% of these samples.

4.2 Finetuning an LLM

One immediate usage for the pool-A data is to finetune the LLM to generate the desired intermediate outputs (correct programs for table QA, or the thought chains) rather than the final answers, which were used as labels in the prior works so far. This approach aligns with the initial reason we need the intermediate data in the first place - on these tasks, it is challenging for the LLM to generate the final answer directly. We have gathered 4800 train samples for the WikiTQ dataset and 5000 samples for the TabFact, with the Binder algorithm. The CodeLlama-34b model was finetuned using the LoRA extension [Hu et al., 2022] with parameters $rank = 64$, $\alpha = 16$. The model has trained for 20k steps till convergence. We witness the accuracy improvement by 3.1% on the test data of WikiTQ in Table 2. Additional data is available in the ablation study (Figure 5). Unfortunately, we were not able to repeat the process for the ReAcTable algorithm, as the LLM used is this case is proprietary.

4.3 Computational resources

For inference with Codellama-34b and Mixtral-8x7b-instruct models we have used eight A100 80GB GPUs by NVIDIA, on internal server with 1280GB of RAM memory, loading them in bf16 precision. The same resources were used for the training of Codellama-34b. We should mention that the full research project required more compute than the experiments reported in the paper, in order to develop the method.

4.4 Results on Table QA task

The application of our method to Binder and ReAcTable solutions for the two Table QA datasets yields the results presented in Table 2. Unfortunately, the authors of ReAcTable have only released the data and the few-shot examples for the WikiTQ dataset, thus the results on TabFact are not reported for this algorithm. Due to using different LLMs, and possibly other factors, we were not able to fully reproduce the accuracy reported in the original papers. However, we have shown the merits of our method by comparing to the reproduced baseline, having all factors equal except the inference protocol. With Binder algorithm, CodeLlama-34b has shown a result close to the reported Codex performance (58.9% accuracy on WikiTQ, instead of the published 64.6%; a similar gap is observed for TabFact). With ReAcTable, the gap is unfortunately much larger (22.4%); this may be attributed

Table 2: Results of Binder and ReAcTable on two TableQA datasets, WikiTQ and TabFact. The reproduction was done using the GitHub repositories of the relevant methods. The original results using Codex were not reproducible. We show significant gains due to our algorithm for both methods. * "FT" stands for finetuned LLM.

Algorithm	Comments	LLM	WikiTQ	TabFact
Binder	published by authors	Codex	64.6%	85.1%
Binder	reproduced by us	CodeLlama-34b	58.9%	78.6%
Binder+ ADLR	Ours	CodeLlama-34b	61.2%	80.6%
Binder+ ADLR	Ours	CodeLlama-34b-FT*	64.3%	83.8%
ReAcTable	published by authors	Codex	68.0%	
ReAcTable	reproduced by us	GPT-3.5-Turbo	62.5%	
ReAcTable	reproduced by us	CodeLlama-34b	45.6%	
ReAcTable + ADLR	Ours	GPT-3.5-Turbo	65.3%	

Table 3: Results on the TabMWP dataset for various configurations. Q-type / A-type mean restriction to examples which questions / answers of the same type as the query.

Sample Selection	# Shots	# Attempts	CodeLlama34b	Mixtral-8x7b-instruct
No restriction	10	5	65.3%	76.4%
Q-type +A-type	10	5	67.3%	83.3%
Q-type +A-type	10	11	70.1%	84.4%
Q-type +A-type	20	11	72.1%	83.9%
Q-type +A-type +ADLR	20	11	77.6%	86.5%

to the more complex instructions and the Program-of-Thought chains employed by the algorithm. So one conclusion is - while with a “smart” enough LLM, ReAcTable achieves better performance, if money is a factor, and using an open-source model is desired (possibly for other reasons, too), Binder is actually better. Our ADLR enhancement helps increase the accuracy by 2.0% - 2.8%.

4.5 Results on the TabMWP dataset

Following [Lu et al., 2022], we study the performance of LLMs on the TabMWP dataset with different sample selection policies. Since the training samples include step-by-step thinking, we interpret the available data as the “Pool A” and apply Part II of ADLR method to find a good subset of those. Results in Table 3 exhibit an ablation study on the context size and the number of LLM runs for each sample, and also exhibit the gain by ADLR, compared to basic selection policies. We find it to be 5.5% and 2.6% for the CodeLlama-34b and the Mixtral-8x7b-instruct models, respectively. The accuracy obtained here is indirectly comparable to the results with the PromptPG algorithm [Lu et al., 2022] with RL-trained sample selection model. They attain 68.2% accuracy using the GPT-3 model in 2-shot ICL regime.

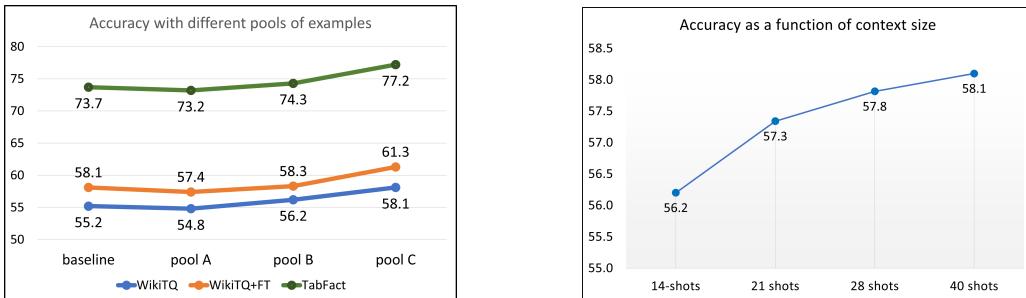


Figure 5: **Left:** performance with different pools of examples. For WikiTQ dataset, we compare the vanilla and the finetuned CodeLlama-34b models (WikiTQ and WikiTQ-FT in the legend). **Right:** Performance as a number of shots for ADLR, compared to the baseline Binder (fixed context)

5 Ablation study

We now study different aspects of our algorithm, as well as the parameters of inference protocols of LLMs in general. In this study, we use the baseline algorithm where the demonstrations contain plain programs code as the requested response, without the Neural API of Binder or the recurrent application of ReAcTAbler.

Refinement phases of context examples In Figure 5 (left) we present the performance levels obtained with different pools of the harvested examples. The baseline computation was done with the 14 plain-code hand-crafted examples provided by Binder authors. We observe that using the entire pool A actually decreases the performance, but restricting it to the hard (interesting) examples, pool B, and further to one-shot useful examples, pool C, helps gaining a significant advantage. Additionally, the accuracy is computed with our finetuned version of the CodeLlama-34b; it shows improvement of 2 – 3% across the pools. The baseline pool is the original 14 hand-crafted examples provided by authors of Binder [Cheng et al., 2023]. In all experiments, we use 20 runs of the LLM per sample.

Context size One of the features unlocked by the ADLR is to enlarge the context size, originally constrained by the number of manually prepared examples. In table 3 the ablation study of the # Shots parameter has shown mixed results - using a context of size 20 instead of 10 improved the performance for CodeLlama-34b, but - interestingly - decreased the accuracy of the (stronger) Mixtral model. In the right part of Figure 5 we present accuracy as a function of # Shots on WikiTQ. Here the performance improves, as expected. The last step of passing from 28 to 40 shots, while maintaining 16k tokens restriction of the input length, was possible thanks to using a more compact representation of the table in the query (we provide details in the Section C.2 of Supplementary material).

Similarity-based shots selection Following [Liu et al., 2022b], we have also attempted to use adaptive retrieval of the few-shot samples using semantic similarity, as was detailed in Section B. However, this method didn't prove useful in our case. First, it comes at cost of using a single context (with K-NN-based shots selection) rather than multiple different ones; second, from the one-shots experiments we had the ground truth about which shots are the "correct" ones for a given query (those that solve it in one-shot prompt). It allows us to observe the distribution of such correct shots in the list of all shots, ranked by the similarity. Unfortunately, we don't see the desired bias of the "good" shots towards the head of the list, which means this metric correlates poorly with usefulness of the shots. More details on this study are given in the Supplementary material Section B. Quantitatively, executing the single K-NN based prompt 20 times with LLM temperature 0.5 yields accuracy of 58.3% on WikiTQ, compared to the 61.2% obtained with 20 randomly sampled prompts; similarly, on TabFact the performance decreases from 80.6% to 75.9%. We conclude that in our case the K-NN method is suboptimal.

6 Afterword

Summary and future work In this work, we have presented a generic method for augmentation of In-Context Learning of LLMs, for the scenario where hand-crafted demonstrations are required. The method is independent of the given task and algorithm for its solution, and seeks to better exploit the weakly labeled data available for the task. This method can be easily adapted to enhance and improve both proprietary and, especially, open source models, where each incremental enhancement is beneficial and crucial, particularly in environments that rely on existing data and hardware. Further exploration of this approach for other tasks, requiring intermediate data, may help us in developing the method and make it more general, thereby inspiring continuous innovation and collaboration within the community.

Limitations The method applies to a specific domain of LLM applications, namely those requiring generation of the intermediate data, such as CoT and executable code programs.

Potential positive and negative societal impacts of our work This work promotes usage of large language models (LLMs) in context of complex question answering. While in general this technology is considered to have positive impact in the society by providing useful service satisfying many needs, it can also have a negative side, if used for producing false textual or visual information. Bias Amplification and Privacy are the main concerns related to LLMs; the usage of our method is therefore conditioned on using well curated and legal datasets.

References

- Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. 2021. On the Opportunities and Risks of Foundation Models. (2021), 1–214. <http://arxiv.org/abs/2108.07258>
- Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems* 2020-Decem (2020).
- Ting Yun Chang and Robin Jia. 2023. Data Curation Alone Can Stabilize In-context Learning. *Proceedings of the Annual Meeting of the Association for Computational Linguistics* 1 (2023), 8123–8144. <https://doi.org/10.18653/v1/2023.acl-long.452>
- Wenhu Chen, Hongmin Wang, Jianshu Chen, Yunkai Zhang, Hong Wang, Shiyang Li, Xiyou Zhou, and William Yang Wang. 2020. Tabfact: a Large-Scale Dataset for Table-Based Fact Verification. In *8th International Conference on Learning Representations, ICLR 2020*.
- Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, Noah A Smith, and Tao Yu. 2023. Binding Language Models in Symbolic Languages. *ICLR* (2023), 1–27. <http://arxiv.org/abs/2210.02875>
- Tianyu Gao, Adam Fisch, and Danqi Chen. 2021. Making pre-trained language models better few-shot learners. *ACL-IJCNLP 2021 - 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, Proceedings of the Conference* (2021), 3816–3830. <https://doi.org/10.18653/v1/2021.acl-long.295>
- Tanmay Gupta and Aniruddha Kembhavi. 2023. Visual Programming: Compositional visual reasoning without training. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition 2023-June* (2023), 14953–14962. <https://doi.org/10.1109/CVPR52729.2023.01436>
- Edward Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. Lora: Low-Rank Adaptation of Large Language Models. *ICLR 2022 - 10th International Conference on Learning Representations* (2022), 1–26.
- Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand,

Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of Experts. (2024). <http://arxiv.org/abs/2401.04088>

Zhengbao Jiang, Yi Mao, Pengcheng He, Graham Neubig, and Weizhu Chen. 2022. OmniTab: Pretraining with Natural and Synthetic Data for Few-shot Table-based Question Answering. *NAACL 2022 - 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference* (2022), 932–942. <https://doi.org/10.18653/v1/2022.naacl-main.68>

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. *Advances In Neural Information Processing Systems* (2012), 1–9. <https://doi.org/10.1162/j.protcy.2014.09.007>

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen Tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in Neural Information Processing Systems* 2020-Decem (2020).

Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2022b. What Makes Good In-Context Examples for GPT-3? *DeeLIO 2022 - Deep Learning Inside Out: 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures, Proceedings of the Workshop 3* (2022), 100–114. <https://doi.org/10.18653/v1/2022.deelio-1.10>

Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian Guang Lou. 2022a. Tapex: Table Pre-Training Via Learning a Neural Sql Executor. *ICLR 2022 - 10th International Conference on Learning Representations Mlm* (2022), 1–19.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-Play Compositional Reasoning with Large Language Models. *NeurIPS* (2023), 1–32. <http://arxiv.org/abs/2304.09842>

Pan Lu, Liang Qiu, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, Tanmay Rajpurohit, Peter Clark, and Ashwin Kalyan. 2022. Dynamic Prompt Learning via Policy Gradient for Semi-structured Mathematical Reasoning. (2022), 1–26. <http://arxiv.org/abs/2209.14610>

Ramesh Nallapati, Bowen Zhou, Cicero dos Santos, Çaglar Gulçehre, and Bing Xiang. 2016. Abstractive text summarization using sequence-to-sequence RNNs and beyond. *CoNLL 2016 - 20th SIGNLL Conference on Computational Natural Language Learning, Proceedings* (2016), 280–290. <https://doi.org/10.18653/v1/k16-1028>

Ansong Ni, Srinivas Iyer, Dragomir Radev, Ves Stoyanov, Wen Tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. LEVER: Learning to Verify Language-to-Code Generation with Execution. In *Proceedings of Machine Learning Research*, Vol. 202. 26106–26128.

Panupong Pasupat and Percy Liang. 2015. Computational Semantic Parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 1470–1480.

Sohan Patnaik, Heril Changwal, Milan Aggarwal, Sumit Bhatia, Yaman Kumar, and Balaji Krishnamurthy. 2024. CABINET: Content Relevance based Noise Reduction for Table Question Answering. In *ICLR*. <http://arxiv.org/abs/2402.01155>

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* 2016-Decem (2016), 779–788. <https://doi.org/10.1109/CVPR.2016.91>

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence embeddings using siamese BERT networks. *EMNLP-IJCNLP 2019 - 2019 Conference on Empirical Methods in Natural Language Processing and 9th International Joint Conference on Natural Language Processing, Proceedings of the Conference* (2019), 3982–3992. <https://doi.org/10.1162/jcnlp/v1/d19-1410>

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémie Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. (2023), 1–48. <http://arxiv.org/abs/2308.12950>

Ohad Rubin, Jonathan Herzig, and Jonathan Berant. 2022. Learning To Retrieve Prompts for In-Context Learning. *NAACL 2022 - 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference* (2022), 2655–2671. <https://doi.org/10.18653/v1/2022.nacl-main.191>

Alexander Scarlatos and Andrew Lan. 2023. RetICL: Sequential Retrieval of In-Context Examples with Reinforcement Learning. *ArXiv* (2023). <http://arxiv.org/abs/2305.14502>

Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. ViperGPT: Visual Inference via Python Execution for Reasoning. *Proceedings of the IEEE International Conference on Computer Vision* (2023), 11854–11864. <https://doi.org/10.1109/ICCV51070.2023.01092>

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* 2017-Decem, Nips (2017), 5999–6009.

Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Koray Kavukcuoglu, and Daan Wierstra. 2016. Matching networks for one shot learning. *Advances in Neural Information Processing Systems* Nips (2016), 3637–3645.

Zilong Wang, Hao Zhang, Chun-Liang Li, Julian Martin Eisenschlos, Vincent Perot, Zifeng Wang, Lesly Miculicich, Yasuhisa Fujii, Jingbo Shang, Chen-Yu Lee, and Tomas Pfister. 2024. Chain-of-Table: Evolving Tables in the Reasoning Chain for Table Understanding. In *ICLR*. 1–24. <http://arxiv.org/abs/2401.04398>

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. SELF-CONSISTENCY IMPROVES CHAIN OF THOUGHT REASONING IN LANGUAGE MODELS. In *ICLR*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv:2210.03629* (2022), 1–33. <http://arxiv.org/abs/2210.03629>

Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. 2023. Large Language Models are Versatile Decomposers: Decompose Evidence and Questions for Table-based Reasoning. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 174–184. <https://github.com/AlibabaResearch/DAMO-ConvAI>

Yiming Zhang, Shi Feng, and Chenhao Tan. 2022a. Active Example Selection for In-Context Learning. *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022* (2022), 9134–9148. <https://doi.org/10.18653/v1/2022.emnlp-main.622>

Yunjia Zhang and et al. Henkel. 2023. ReAcTable: Enhancing ReAct for Table Question Answering. In *arXiv:2310.00815[cs]. PVLDB*. <https://arxiv.org/abs/2310.00815>

Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022b. Automatic Chain of Thought Prompting in Large Language Models. (2022). <http://arxiv.org/abs/2210.03493>

Yilun Zhao, Linyong Nan, Zhenting Qi, Rui Zhang, and Dragomir Radev. 2022. REASTAP: Injecting Table Reasoning Skills During Pre-training via Synthetic Reasoning Examples. *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022* (2022), 9006–9018. <https://doi.org/10.18653/v1/2022.emnlp-main.615>

A Reproducibility

ADLR code is based on Open Source (OS) releases of the BInder and ReAcTable codebases. We submit the code representing a fork of the github repo for the ReAcTable algorithm, with some modifications and additional files implementing our method. Our code will be released as an Open Source github contribution upon acceptance.

B Evaluation of the K-NN sample selection based on semantic similarity

As mentioned in Section 5, we have studied the relevance of the semantic similarity (of the candidate examples to the query), to the success of the LLM on this query when using these examples in the context. Specifically, we used the model **all-mpnet-base-v2** of type SentenceTransformers [Reimers and Gurevych, 2019] to compute embeddings of the samples, where each sample is represented by the text string of the form [(table headers):(one table row):(question)]. The embeddings are unit-norm vectors, so we compute the cosine similarity between the query and each pool example to get ranking of the pool with respect to the given query.

In second stage of part II of our method, we are executing one-shot inference, where the shots come from pool B of solved examples, and the queries are from the set of difficult samples. Thus for each query sample we get the list of shots which helped the LLM solve it (the "good shots"). Now, combining this information with the ranking of the same pool of shots by the semantic similarity, we obtain the distribution of the good shots in the entire pool; we plot the obtained distributions for random four queries in Figure 6. Unfortunately, the desired bias of the distributions towards the head of the list (left) is not observed; this accounts for the fact the performance on the test set was not improved by using the K-NN based approach.

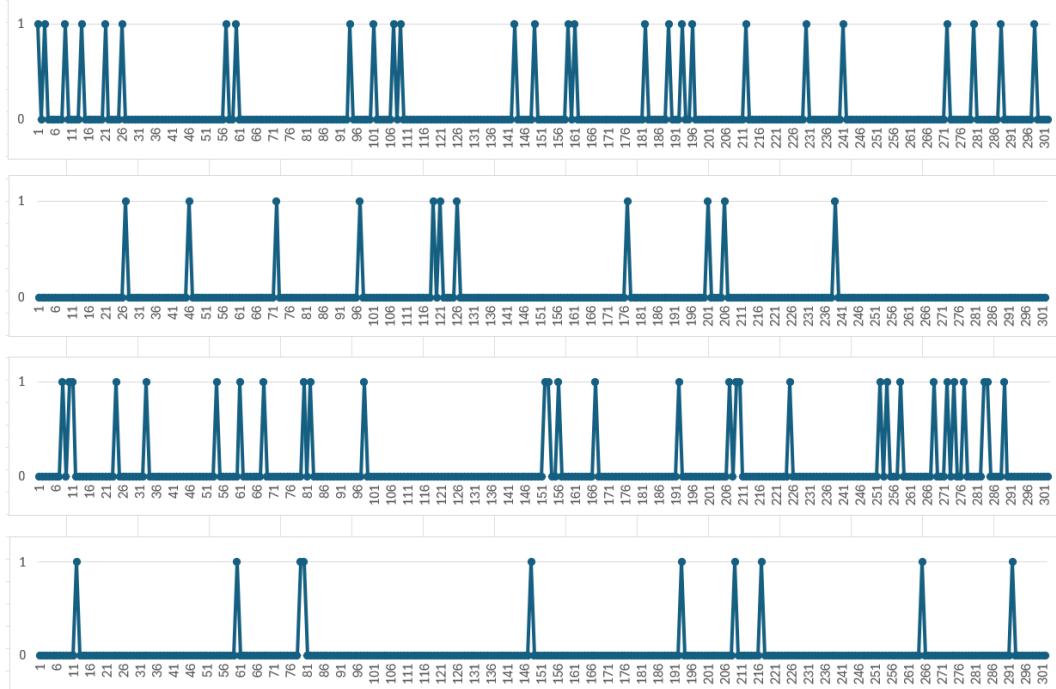


Figure 6: Distributions of the good shots in the similarity-ranked list of the pool examples, for a few queries

```

CREATE TABLE Ireland in the Eurovision Song Contest 1990(
    row_id int,
    draw int,
    artist text,
    song text,
    points int,
    place text)
/*
row_id  draw   artist  song           points  place
0      1     the memories "if it means losing you"  57  8th
1      2     ann breen   "oh, darling"        80  4th
2      3     fran meen   "say that you love me"    66  6th
*/
Q: what song received the most points?
Python:
def solve(table):
    sub_table = table
    tmp = [(x, y) for x, y in zip(sub_table['song'], sub_table['points'])]
    tmp = sorted(tmp, key=lambda x: x[1], reverse=True)
    result = list(map(lambda x: x[0], tmp))[0]
    return result

CREATE TABLE 1950-51 in Scottish football(
    row_id int,
    date text,
    venue text,
    opponents text,
    score text,
    competition text,
    scotland scorer(s) text)
/*
row_id date   venue  opponents  score  competition scotland scorer(s)
0  1950-10-21 00:00:00 ninian park, cardiff (a)  wales  3-1 bhc lawrie reilly (2), billy liddell
1  1950-11-01 00:00:00 hampden park, glasgow (h)  northern ireland  6-1 bhc billy steel (4), john mcphail (2)
2  1950-12-13 00:00:00 hampden park, glasgow (h)  austria 0-1 friendly  none
3  1951-04-14 00:00:00 wembley stadium, london (a)  england 3-2 bhc bobby johnstone, lawrie reilly, billy liddell
4  1951-05-12 00:00:00 hampden park, glasgow (h)  denmark 3-1 friendly  billy steel, lawrie reilly, robert mitchell
5  1951-05-16 00:00:00 hampden park, glasgow (h)  france 1-0 friendly  lawrie reilly
*/
Q: how many goals did the scotland national team score during the 1950 british home championship?
Python:
def solve(table):
    import datetime
    import re
    sub_table = table.query(`competition` .str.find("bhc")>=0')
    sub_table['date'] = pd.to_datetime(sub_table['date'])
    sub_table['year'] = sub_table['date'].apply(lambda x: x.year)
    sub_table['month'] = sub_table['date'].apply(lambda x: x.month)
    sub_table['day'] = sub_table['date'].apply(lambda x: x.day)
    sub_table['score'] = sub_table['score'].apply(lambda x: re.split('-', x)[0])
    sub_table['score'] = pd.to_numeric(sub_table['score'])
    sub_table = sub_table.query(`year`==1950')
    result = sub_table['score'].sum()
    return result

```

Figure 7: Examples of train samples with automatically generated code

C Data and context examples for the relevant datasets and algorithms

C.1 WikiTQ - example of automatic annotations

We provide some examples of the train samples with automatically generated python code required for the context demonstrations. These are given in Figure 7. The text is structured by the \t, \n markers, so it is actually given as a single prompt-ready text string. In the displayed examples one can learn about the coding style of CodeLlama-34b, albeit influenced (strongly) by the original hand-made examples. In the second example, the operations are broken into small steps, with some repeatability a human programmer might replace with a loop over different headers.

C.2 WikiTQ - Compact table representation in context

We have found it beneficial to enlarge the number of shots in the context, even at expense of reducing the size of each example to fit into the prompt size restriction (and also because LLMs loose track of information in very long prompts). Specifically, we begin with the format in which the context examples were presented by the Binder authors (Figure 8. Specifically, we have [1] removed the table header and the column types, [2] removed the /*, ..., */ brackets and the

"3 example rows: SELECT * FROM w LIMIT 3" text, repeating in every example, and [3] retained only one table row instead of three. The question and the desired output (program) were retained in full.

```

CREATE TABLE 2007 New Orleans Saints season(
    row_id int,
    week int,
    date text,
    opponent text,
    time text,
    game site text,
    tv text,
    result/score text,
    record text)
/*
3 example rows:
SELECT * FROM w LIMIT 3;
row_id  week  date  opponent  time  game site  tv  result/score  record
0 1  2007-9-6  indianapolis colts  t20:30 edt  rca dome  nbc 1 41 - 10  0-1
1 2  2007-9-16  tampa bay buccaneers  t13:0 edt  raymond james stadium  fox 1 31 - 14  0-2
2 3  2007-9-24  tennessee titans  t20:30 edt  louisiana superdome espn  1 31 - 14  0-3
*/
Q: what number of games were lost at home?
Python:
def solve(table: pd.DataFrame):
    sub_table = table[(table['result/score'].split(' ')[0]< table['result/score'].split(' ')[2]) & (table['opponent'].split(' ')[0]=='New') &
                      (table['opponent'].split(' ')[1]=='Orleans')]
    result = len(sub_table)
    return result
headers: row_id week  date  opponent  time  game site  tv  result/score  record
values: 0 1  2007-9-6  indianapolis colts  t20:30 edt  rca dome  nbc 1 41 - 10  0-1
Q: what number of games were lost at home?
Python:
def solve(table: pd.DataFrame):
    sub_table = table[(table['result/score'].split(' ')[0]< table['result/score'].split(' ')[2]) & (table['opponent'].split(' ')[0]=='New') &
                      (table['opponent'].split(' ')[1]=='Orleans')]
    result = len(sub_table)
    return result

```

Figure 8: Top: full context example from WikiTQ dataset, in Binder format; bottom: the same example in our compact format

C.3 TabMWP

The training part of the TabMWP dataset contains 23k samples, each representing a table, a question (mathematical problem) involving the table data, a reasoning step-by-step process (text), and finally the answer. Each samples is classified according to a question type, answer type and grade, as detailed in Table 4.

Table 4: Types of questions and answers in the tabMWP dataset

Name	Amount in training data	
Question type	free_text	17315
Question type	multi_choice	5744
Answer type	integer_number	13804
Answer type	extractive_text	2991
Answer type	decimal_number	3511
Answer type	boolean_text	2467
Answer type	other_text	286
grades	1-8	

As was shown in the ablation study (Table 3 in the paper), the LLM benefits from receiving in-context examples with question/answer of same type as a query. A few examples of the data are provided in Table 5:

Table 5: Examples from the tabMWP dataset

Table	Question	Solution	Choices	Answer														
<table> <thead> <tr> <th>Stem</th> <th>Leaf</th> </tr> </thead> <tbody> <tr> <td>3</td> <td>3, 3, 3, 5, 5</td> </tr> <tr> <td>4</td> <td>6</td> </tr> <tr> <td>5</td> <td>4, 5, 7, 8</td> </tr> <tr> <td>6</td> <td>7, 8</td> </tr> <tr> <td>7</td> <td>2, 3, 7, 9</td> </tr> <tr> <td>8</td> <td>6, 8, 9</td> </tr> </tbody> </table>	Stem	Leaf	3	3, 3, 3, 5, 5	4	6	5	4, 5, 7, 8	6	7, 8	7	2, 3, 7, 9	8	6, 8, 9	The members of the local garden club tallied the number of plants in each person's garden. How many gardens have at least 47 plants?	Find the row with stem 4. Count all the leaves greater than or equal to 7. Count all the leaves in the rows with stems 5, 6, 7, and 8. You counted 13 leaves, which are blue in the stem-and-leaf plots above. 13 gardens have at least 47 plants.	NA	13
Stem	Leaf																	
3	3, 3, 3, 5, 5																	
4	6																	
5	4, 5, 7, 8																	
6	7, 8																	
7	2, 3, 7, 9																	
8	6, 8, 9																	
<table> <thead> <tr> <th>Day</th> <th>Tickets</th> </tr> </thead> <tbody> <tr> <td>Friday</td> <td>71</td> </tr> <tr> <td>Saturday</td> <td>74</td> </tr> <tr> <td>Sunday</td> <td>75</td> </tr> <tr> <td>Monday</td> <td>72</td> </tr> </tbody> </table>	Day	Tickets	Friday	71	Saturday	74	Sunday	75	Monday	72	The transportation company tracked the number of train tickets sold in the past 4 days. On which day were the fewest train tickets sold?	Find the least number in the table. Remember to compare the numbers starting with the highest place value. The least number is 71. Now find the corresponding day. Friday corresponds to 71.	Friday, Saturday, Sunday, Monday	Friday				
Day	Tickets																	
Friday	71																	
Saturday	74																	
Sunday	75																	
Monday	72																	
<table> <thead> <tr> <th>Level</th> <th>Number</th> </tr> </thead> <tbody> <tr> <td>Gold</td> <td>15</td> </tr> <tr> <td>Silver</td> <td>68</td> </tr> <tr> <td>Bronze</td> <td>58</td> </tr> </tbody> </table>	Level	Number	Gold	15	Silver	68	Bronze	58	The Burlington Symphony categorizes its donors as gold, silver, or bronze depending on the amount donated. What fraction of donors are at the bronze level?	Find how many donors are at the bronze level. 58. Find how many donors there are in total. $15 + 68 + 58 = 141$. Divide 58 by 141. $58/141$. $58/141$ of donors are at the bronze level.	NA	58/141						
Level	Number																	
Gold	15																	
Silver	68																	
Bronze	58																	